Detecting Dynamic Security Threats in Multi-Component IoT Systems

Isaac Shrestha
University of Nebraska at Omaha
ishrestha@unomaha.edu

Matthew L. Hale
University of Nebraska at Omaha
mlhale@unomaha.edu

Abstract

The rising ubiquity of the Internet of Things (IoT) has heralded a new era of increasingly prolific and damaging IoT-centric security threat vectors. Fast-paced market demand for multi-featured IoT products urge companies, and their software engineers, to bring products to market quickly, often at the cost of security. Lack of proper security threat analysis tooling during development, testing, and release cycles exacerbate security concerns. In this paper, we augment a security threat analysis tool to use audit hooks, open-source information capture components, and machine learning techniques to profile dynamic wearable and IoT operations spanning multiple components during execution. Our tool encourages data-drive threat identification and analysis approaches that can help software engineers perform dynamic testing and threat analysis to mitigate code-level vulnerabilities that lead to attacks in IoT applications. Our approach is evaluated by means of a case study involving a system evaluation across several common attack vectors.

1. Introduction

Internet of Things (IoT) is a term that both refers to actual devices (i.e. "things" on an internet-connected network) and an architectural paradigm. Things in the IoT may be as small as a single sensor in a toaster that sends an alert when the toast is ready to automobile components that collect, analyze, and make decisions on real-time driving data, to wearable watches and other devices that capture human activities for bio-feedback and exercise tracking purposes. Depending on the commercial, medical, and personal use cases, data collected by IoT devices may be particularly personal (e.g. pacemaker or glucometer logs) [1] or highly sensitive (e.g. GPS coordinates that could be used to track user movements) [2]. For instance, in a widely reported side-channel, GPS coordinates from the Strava fitness wearables on military service members made its way onto world maps, inadvertently revealing the location and commonly used walking pathways of several secret military installations [3].

As IoT devices have proliferated into the wider internet to the tune of nearly 10 billion devices [4], the risks of misuse have also undoubtedly increased in diversity and likelihood. This comports with the increasing number of high profile attacks, such as the Mirai Botnet [5] which briefly shut down access to a large swath of high traffic websites by coopting and misusing unprotected digital video recorders (DVRs) owned by unsuspecting users, and with academic and industry studies of the IoT threat landscape [6]. These studies have highlighted several features of wearables and, more broadly, IoT, such as sensing and communication capabilities, always-on network connectivity, and pervasive embedded locations, that make them particularly vulnerable to attack.

As with other software and hardware design and development, security in IoT systems is often an afterthought [7] to getting products to market. The problem is exacerbated by the paradigmatic assumptions of IoT – which emphasize low power hardware, plug-and-play modular design, and low or no configuration. While many threat analysis and evaluative testing tools exist for web and network-based penetration testing [8-10], IoT designers and developers face a lack of robust evaluative tooling for examining potential threats that span more than one component of their product architecture (e.g. an attack that involves a chain of events from the network, a phone, an IoT device, and a web site).

This lack of parity between traditional network and web penetration testing tooling (e.g. Wireshark [8], Nmap [9] Metasploit [10], etc.), and tooling available for pen-testing IoT hardware, Bluetooth networks, and mobile app analysis has largely translated to less time spent on IoT product security evaluations [6].

This paper develops and documents a set of audit hooks and data capture mechanisms for accumulating threat intelligence from multiple sources and perspectives (e.g. app data, Bluetooth data, web traffic, etc) and combining it to identify potential security threats and direct mitigation efforts in IoT. Using these data capture mechanisms, we augment a testbed, created in prior work [11], called *SecuWear* so

URI: https://hdl.handle.net/10125/60151
ISBN: 978-0-9981331-2-6

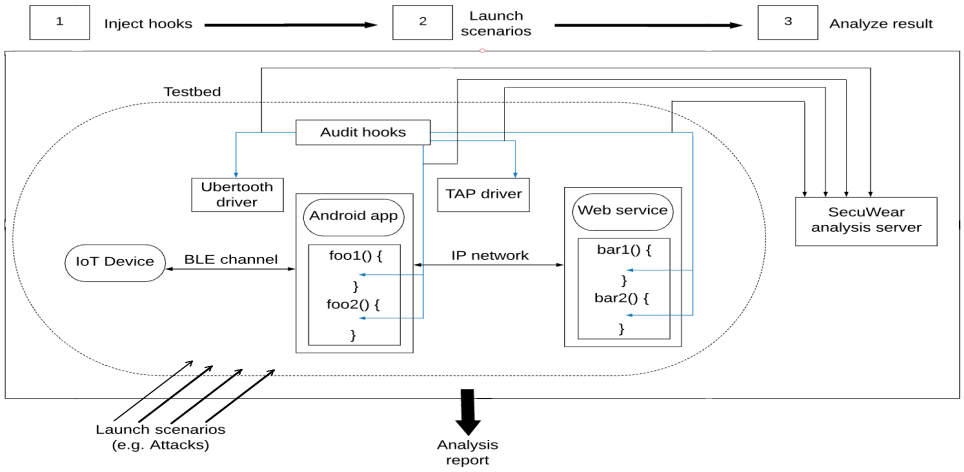

Figure 1. High-level overview of the SecuWear IoT testbed using audit hooks to gather, accumulate and analyze data to identify attack scenarios and direct mitigation efforts.

that it is capable of identifying attacks in IoT systems which include one or more IoT devices, an android-based mobile application, one or more web services, and the communication mediums between them, i.e. Bluetooth / Bluetooth Low Energy (BLE) and IP networks. The testbed design is shown in Figure 1. Here a particular device is communicating with an Android app over BLE, using functions foo1 and foo2, which are subsequently interacting with a RESTful API in a web service using endpoints bar1 and bar2. Audit hooks are placed in each function on the app, endpoint on the webservice, and on passive network listening tools – including an Ubertooth (a low-cost, open source, Bluetooth monitoring hardware device) and a network tap feeding data to WireShark. The design and placement of the audit hooks allow them to operate in real-time as an application executes to capture ephemeral events and internal data that may exist only within the scope of the function and/or for the duration of the process. The testbed gathers events, pools them with other data gathered from other components in the architecture and creates event traces of execution that can be used to profile normalcy, identify potentially adherent behavior that may be associated with known attack vectors, and/or detect new types of attacks. In this sense, SecuWear is a testbed that facilitates dynamic testing during development or monitoring, when an app has been deployed to production. Analysis in SecuWear is app specific since the data gathered may vary between applications. More information about the audit hooks and analysis techniques is provided in Section 3.

The rest of the paper is organized as follows. Section 2 reviews relevant literature. Section 3 describes the design model of the tool, the design decisions underlying the developed audit hooks, the process followed to pull information gained from audit hooks together with other sources of information such as that from Ubertooth, Web servers, and network data. Section 4 demonstrates the feasibility of differentiating between behaviors associated with attack and normal operations of a real-world IoT system. A case study involving an IoT device, an Android App, and an advertisement web server is used in the demonstration. Section 5 concludes the paper.

2. Background

2.1. Security concerns in IoT and wearables

Security weaknesses in IoT arise due to range of root causes related to the functional behavior of the IoT device in question [11, 12], processes which occur during operation [7], and the multiple security domains the devices operation within [13]. Typical IoT devices interact with higher-powered devices such as smart phones and web services, as well as other IoT devices. Interaction often occurs over Bluetooth or BLE, and Wi-Fi. Given the highly integrated architecture and low-powered hardware involved in IoT, the existence of a vulnerability in a connected app, component, or service, can potentially lead to the compromise of the IoT device [14]. Wearables devices are particularly vulnerable due to computational and power limits [12] which limit the implementation of strong encryption and authentication protocols. BLE is susceptible to range of attacks from passive eavesdropping to active interference [7, 15]. Apps and devices are also sources of concern given the wide range of vulnerabilities targeting operating systems such as the published CVE entries for Android [16].

Gegick and Williams [17], suggest that attacks on wearables are often based on small and known attacks, that act as gateways for larger and more sophisticated attacks. Their work highlights the common, often unmaintained, stagnancy of IoT device software – which go unpatched, allowing for vulnerability re-use by attackers. Design flaws are also a large source of concern [18]. These flaws emerge, among other reasons, due to poor consideration of security requirements [18]. No matter how secure software engineers claim their application to be there are always weaknesses that can be exploited to break in. Appropriate evaluative tooling has also been identified as a critical pre-requisite for secure software [12].

2.2. Multi-component threat analysis system

Multi-component threat analysis (MCA) is a system-spanning activity that identifies and profiles potential threats which may span one or more components in the system. IoT applications often incorporate many components and services, making MCA a critical part of overall system evaluation.

Siboni et al. [12] developed a security testbed framework for MCA of wearable applications. Their approach examines a wide range of wearable devices that transmit data through variety of communication channels including WiFi, Zigbee and Bluetooth. Their framework enables security testing based on memory consumption, CPU utilization, and the file system of the device during context-based attacks (when different simulators maliciously trigger internal sensors) and data attacks (when external signals and data manipulate the sensors). It spans components, considers device activities upon identifying a threat, but does not collect or monitor real-time event traces and does not identify application level vulnerabilities.

Intrusion detection, and *intrusion detection systems* (IDS), are related concepts within the scope of MCA which focus, specifically, on the identification of threats in real-time data – often for the purpose of mitigating or preventing threat-actors from conducting network-based attacks [19]. Zhang et al. [20] presents an IDS architecture for examining and preventing attacks in mobile wireless networks. Their architecture is based on an ad-hoc routing protocol which discovers the network topology of a wireless network and then monitors it to build notions of normality. While their approach is useful for detecting anomalies in mobile wireless networks environments, it does not identify threats spanning component boundaries in IoT.

Across MCA research in IoT, much work has focused on network-data [20] spanning components, but little has been done to extract disparate, differently tiered data from the components themselves to analyze the security posture of the overall system [21]. One potential in-roads towards component-based MCA in IoT are *audit hooks*. Audit hooks are small functional insertions that occur within programs and functions that allow components to pass operational information up to an, often central, auditor. Audit hooks have been used and studied extensively within the web services community as a means to examine and track security behavior across service compositions [22]. Here, a central auditor collects information from contributing audit hooks, combines the information to synthesize an analysis to determine if the composition is meeting its service requirements (or *service levels*), and then alerts stakeholders if the constraints are not met [22]. Others in the IoT MCA research community have highlighted audit hooks as options for IoT systems [23], but to our knowledge no frameworks make wide usage of audit hooks to integrate data across multiple disparate components and differing abstractions in IoT architectures (e.g. data arising from functional execution of code on smartphones, connected webservices, and the IoT device hardware).

2.3. Identifying attacks with machine learning

Using machine learning (ML) to identify attacks is not a new concept. The earliest application we found was in 1975, where Carlstedt et al. [17] abstracted system calls, data stores and other entities in operating system source code into generalized patterns and used them to find security flaws. They used concepts such as finite state automata (FSA) to show that sequence of events could result in an attack in practice. Graphical structures such as attack trees, attack nets were also proposed for threat identification.

More advanced ML approaches including classifiers [20, 24] have been used by intrusion detection systems to differentiate between different types of traffic according to signatures of behavior. For instance, Zhang, Lee, and Huang, [20] apply two classification models: (i) RIPPER and (ii) SVM-light to separate data into attack or non-attack classes in mobile ad-hoc networks. Their approach relies on attack patterns to characterize and train their models. The efficacy of attack patterns and model-based threat detection is predicated on the notion that normal and intrusion activities have distinct and detectable behaviors [20]. Attack patterns, in this sense, are class attributes (features) that exist during attack scenarios. ML algorithms utilize these attack patterns and classify data into either one of the classes based on presence or absence of known patterns.

Buczak and Guven [24] survey and explore how various ML approaches can be used for cyber security applications. Among others, they examined

applications of artificial neural networks, Bayesian networks, support vector machine, and decision trees for classification of security data into classes associated with attack patterns. For each approach, they provide and apply typical ML comparison criteria such accuracy, positive prediction value (or precision), sensitivity, and specificity, to security problems. Their work highlights the role of ML for security problems such as anomaly detection and misuse case detection. In addition, they identify the following steps in using ML for security-related data:

- Clean and prepare data into test and training sets.
- Identify class attributes (features) from the training set to characterize attack patterns.
- Identify the subset of attributes necessary for classification (i.e. dimensionality reduction).
- Train the model on the features.
- Using trained model to classify samples in test set.
- Measure and evaluate model.

3. Audit Hooks for Dynamic IoT Testing

The design of the SecuWear testbed makes extensive use of developer-customizable audit hooks inserted into different method, components, and network channels within the architecture of an IoT app. The audit hooks serve to intercept function calls, messages, or events passed between software components and must exist within and across components to capture interactions and transmissions as they occur in IoT devices or wearables and collaborating smartphones and web services. The audit hooks can operate on a real-time basis and capture data that might exist only in an internal form or only for the duration of the process. Appropriate placement of audit hooks is critical to detecting anomalies. Once events are captured, they are fed into a centralized server. The server combines events gathered from multiple perspectives (e.g. function data, device data, packet data, etc) and compiles it into an event trace to prepare it for analysis, before finally producing a vulnerability analysis report with any findings. The schema for events in the trace is shown in Figure 2.

```
Eventitem:
      created: datetime
      domain: string
      eventtype: string
      event: string
      codereference: string
      data: string
      size: string
      run: integer
```

Figure 2. Schema used for events in the trace

Each event item in the trace includes the fields: *created* which is a unix timestamp identifying when the event was captured; *domain* which identifies where the event originated (can take on the discrete values such as IoT device, mobile app, web or identifiers for the underlying networks); *eventtype* is a high level category of events which exist within a domains (e.g. “function execution” or “error handler”); *event* is the specific type of event that exists within the eventtype category (e.g. “foo invoked”); *codereference* is the specific line of code where the event originated; *data* is a field which allows for more specific information such as an error log or I/O data related to the event to be logged (e.g. packet information on a network-based event); *size* is the size of data and is mostly used for binary data (such as network traffic); finally, *run* is the event trace the event item exists within.

In this section, we discuss the design decisions that led to the creation of the audit hooks, the process the audit hooks and analysis server follow to pool, aggregate, and synthesize event traces, and the specific series of steps developers must follow to use the audit hooks and testbed to analyze an application of interest.

3.1. Audit hook requirements

Several requirements guided the creation of the audit hooks: (1) they must be applicable across differing apps, (2) they must be useful regardless of the IoT/wearable device being evaluated, (3) they must capture a range of information to characterize behavior across the entirety of an IoT system architecture, and (4) they must support event-trace analysis techniques.

3.2. Designing the hooks

To meet these requirements, we created three separate types of audit hooks, one for passive Bluetooth network taps, one for smartphone apps (particularly for usage in the functions interacting with wearable or IoT hardware), and one for insertion into any potentially collaborating web services. Each hook type is reviewed below.

The first type of hook, for gathering Bluetooth data, sits on top of the open-source Ubertooth project. Ubertooth comes out of the box with an embedded driver and command line interface (CLI) capable of capturing packets on Bluetooth and BLE networks. Captured packets are converted automatically by the Ubertooth firmware into a Wireshark-ready format (PCAP) that is typical of network traffic analysis. Our audit hook intercedes on the Ubertooth firmware to collect and send captured packet data to the SecuWear aggregation server (where other event data is

gathered). This particular hook was developed using a Python library called PyShark [25]. Ubertooth captures data including the advertising header of the packet, mac addresses for source and target, BLE protocol type, and the specific data for transmission. These data broadly and narrowly characterize the communication behaviors of an IoT system of interest. Packet data is widely researched, so alone, this is not a novel invention. The novelty of using Ubertooth data is in its combination with data from other audit hooks.

The second kind of audit hook is useful within apps on smartphones exchanging information with a wearable or IoT device. This kind of hook characterizes the specific temporal ordering of functional execution and data operations on the phone related to data from the wearable or IoT device. This type of hook, referred to henceforth by its classname AppHook was developed for Android using the open-source *OkHttp* [26] library. OkHttp provides utilities for making HTTP requests and handling the responses. AppHooks allow software developers to instantiate the event schema in Figure 2 with data specific to their application, such as data arising from functional execution in a function of interest in their architectural design. The AppHook class, shown in Figure 3, encapsulates event logging to the SecuWear server.

To use an AppHook, a developer needs only to import the class, decide where they wish to log information in their app, instantiate the AppHook class by calling its constructor (lines 12-14) and passing it the url where the SecuWear aggregation server is running, and then call the *logEvent* method (lines 16-31 in Figure 3) at the chosen location, passing it the information they wish to log. In most cases the SecuWear testbed is run locally, meaning the url would be *http://localhost/api/events/*, if the developer wishes to capture information after an app has gone into production, they would host SecuWear online and then use the url where the platform is running. Any information deemed relevant can be passed to logEvent as specified in the schema in Figure 2.

The third type of hook allows developers to log information arising from connected web services, such as internal information communicated to it by the app, or other functions executing on aggregate data on the web server. For web requests, SecuWear provides a web-platform agnostic API endpoint for logging data. The API format conforms to JSONAPI standard [27] and exposes the event schema in Figure 2. The SecuWear event logging API is web service platform agnostic (meaning it works with any webserver of interest). As an example, we implemented a method, using the *Httplib2* [28] library an open-source HTTP client library for Python, which can be used by Python-based webservers, such as Django [29] to log information of interest. The example implementation is shown in Figure 4. This script exemplifies API invocation and would look similar (albeit with different library usage) for other web application server languages.

```
import okhttp3.Call;                                                    1
import okhttp3.Callback;                                                2
import okhttp3.FormBody;                                                3
import okhttp3.Request;                                                 4
import okhttp3.OkHttpClient;                                            5
import okhttp3.RequestBody;                                             6
                                                                        7
public class AppHook {                                                  8
   String secuwear_endpoint = "";                                       9
   OkHttpClient client;                                                10
                                                                       11
   AppHook (String api_url){                                           12
      secuwear_endpoint = api_url;                                     13
      client = new OkHttpClient();                                     14
   }                                                                   15
   public void logEvent(Long time, String e, String type,              16
      String ref, String size, String data)                            17
   {                                                                   18
      RequestBody body = new FormBody.Builder()                        19
        .add("created", time.toString())                               20
        .add("eventtype", type)                                        21
        .add("event", e )                                              22
        .add("codereference", ref)                                     23
        .add("size", size)                                             24
        .add("data", data)                                             25
        .add("domain", "Mobile")                                       26
        .build();                                                      27
                                                                       28
      Request request = new Request.Builder();                         29
      request.url(secuwear_endpoint).post(body).build();               30
   }                                                                   31
}                                                                      32
```

Figure 3. Mobile app audit hook in Android

```
import time                                                             1
import httplib2                                                         2
import urllib                                                           3
                                                                        4
def webHook(url, time, event, type, coderef, size, data):               5
   payload = {                                                          6
      'created': time.time() * 1000, # convert to ms                    7
      'eventtype': type,                                                8
      'event': event,                                                   9
      'codereference': coderef,                                        10
      'size': size,                                                    11
      'data': data,                                                    12
      'domain': "WebApp",                                              13
   }                                                                   14
   body = urllib.urlencode(payload)                                    15
   headers = {'Content-type': 'application/x-www-form-                 16
              urlencoded'}                                             17
   channel = httlib2.Http()                                            18
   channel.request(url, headers, method="POST", body)                  19
```

Figure 4. Code snippet of web audit hook

All three hook types, i.e. Bluetooth logging, Mobile AppHooks, and web service audit hooks (including API information and the example implementation) have been open sourced and are available on GitHub. Each hook is, respectively, available in [30, 31, 32].

3.3. Using the audit hooks in practice

The audit hooks, when taken together capture data end-to-end across an IoT application architecture. By collecting data from different perspectives, security evaluators and software developers can gain insight into the cross-boundary behaviors of the components

in their application architecture, identify problem areas, and direct their mitigation efforts. To use the testbed, a developer or evaluator needs to inject the hooks, determine and launch attack scenarios of interest on the IoT system, and then use SecuWear to analyze the results to determine if the app was vulnerable to attack or to discover potential issues.

When injecting the hooks, a developer needs to determine *where to place the audit hooks*. Analysis is only as good as the quality, quantity, and coverage of data captured in the system. Often, developers have design documents, software architectures, and risk assessments that can direct placement to components of most concern. If developers are unsure about placement, we recommend injecting a hook into each method that has sensitive or critical data assets.

The second step in the process involves launching scenarios of interest to profile the behavior of the system, as captured by the audit hooks, and construct a model of correct behavior. Once 'normal' operational states are known, the developer can launch various forms of attacks and observe the system behavior to determine if problems exist or the system is vulnerable to attack. In practice, this may mean a series of tests launched as part of the DevOps process during development or it may mean monitoring systems in production to gather more robust test data.

Analyzing the results can be both manual and automated. In this work, we demonstrate a proof-of-concept supervised machine learning technique for training a model on correct and normal system behaviors and malicious abnormal behaviors associated with attacks. We then use the model to determine if attacks have occurred. When attacks are detected in this way, the event traces which led to detection can be recalled by the SecuWear analysis server and presented to the developer or evaluator to direct mitigation efforts to certain components.

3.4. Modeling normalcy with machine learning

As an example of the types of analysis that can occur, we explored supervised machine learning techniques for building classifiers. Here, each model is trained with labeled data of several types including normal operational data observed during the typical operation of the system and operational data gathered when the system is under some form of attack. In this paper, we examined four multi-class classification algorithms: logistic regression, one-vs-one SVM, multinomial Naïve Bayes, and K-NN to test their efficacy for determining non-attacks from attacks.

In practice, a developer or security evaluator could use the classifiers in production or during testing in the DevOps process (e.g. before releasing a build) by launching attack scenarios as shown in Figure 1 to determine if the system is vulnerable to attack. During development this could highlight abnormal behavior that could be patched before release. If used in production, the classifier could serve as a monitor to detect abnormal behavior and alert the developer to the potential for misuse. The specifics of our analysis process are discussed further in the next section.

4. Evaluation

To evaluate the audit hooks in the SecuWear testbed we used a case study. We applied the audit hooks to an example IoT system, gathered data about its operation under attack and normal conditions, and then used the machine learning techniques introduced in Section 3 to analyze the results.

4.1. The case

The case for further study is as follows. A developer is writing code for multi-component IoT system that includes a fitness tracking wearable with accelerometer, barometer, and thermistor sensors – similar to those on a Fitbit. The wearable will send its data to an Android app which will display recent accelerometer and temperature data. Bluetooth LE will be used for communication between the watch and the app. The app will interface with a webservice to store historical temperature and accelerometer data over time for the user to review in aggregate. The Android app will also display advertisements from a third-party advertisement provider service.

The wearable watch will be based on a MetaWearC [33] hardware device, chosen for its open-source API, low cost, and variety of sensors. The MetaWear C has an accelerometer, barometer, and thermistor [33], among other sensors. The developer wants to support all versions of Android above 6.0.1(Marshmallow) to support most of the Android market. Figure 5 shows a component-based system diagram of the proper operational flow expected by the developer in the case. As it shows, the developer has chosen to use an Android WebView [34] to insert web-based advertisements (i.e. HTML, JavaScript, CSS).

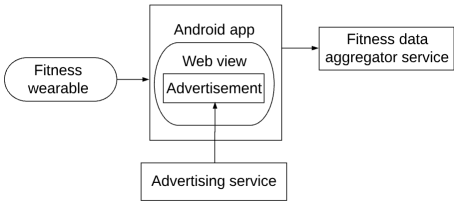

Figure 5: Component-based system diagram and expected data flow of the case system

4.2. Attack scenarios in the case

The developer wants to evaluate the system for potential security vulnerabilities. In this case, the developed system is vulnerable to two attacks. The first scenario involves a malicious advertisement that has made it onto the third-party advertising service. Similar attacks have been described in [35]. The advertisement infects devices it is displayed on by injecting a short JavaScript script into any affected devices to collect all exposed variables within its scope and periodically read and send their values to a malicious logging server. This is an example of *data exfiltration*. The system vulnerability which allows for this attack, called *code injection*, to occur is that the system does not properly escape scripts, allowing code to be loaded and executed at run time. The code injection attack scenario is shown in Figure 6.

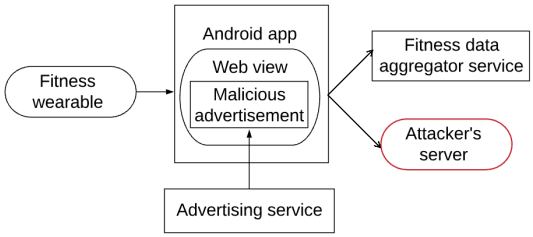

Figure 6: Code-injection attack allows malicious JavaScript to exfiltrate sensor data to an attacker's server.

The second attack involves misuse of permissions. In this case, the system exposes all of its variables to the advertisement web view using an Android JavaScript interface. This means that the WebView component in the app has access to the sensor data from the wearable, the state variables which identify the url where the fitness aggregator service exists, and other data in the program. This type of poor permission handling shows that the app has not implemented *least privilege* and has given the code controlling the advertisement integration more privileges than it needs allowing it to read and write to variables such as the sensor values, i.e. the temperature and accelerometer sensor data from the wearable. In this attack the malicious advertisement interrupts communication with the fitness aggregation service by replacing the web service URL with a new URL which points to a malicious server. This drastically affects acts a *denial of service* – since it prevents the correct operation of the app. Other similar attacks might re-write sensor variables affecting *data integrity*. The denial of service attack scenario is shown in Figure 7.

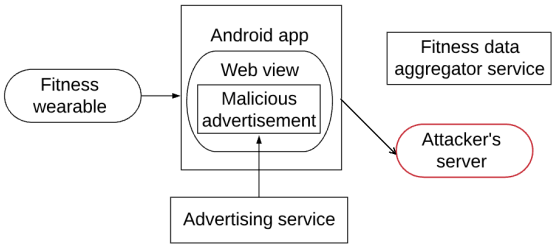

Figure 7: Denial of service and exfiltration attack prevents normal operation while also sending data to the attacker's server.

4.3. Implementation of the case

We implemented the case exactly as described to evaluate SecuWear and the audit hooks. To realize the fitness use cases, we forked the open-source MetaWear Android app [33] on GitHub. This app already has out-of-the-box code to interact with the MetaWearC, pull temperature, accelerometer, and other sensor data from the device, and display it graphically in the app. We modified the app to have a WebView which display ads from an advertisement server. We also modified the app to send its temperature and accelerometer data to a data aggregator service. For the data aggregator service, we setup a simple Django server with two API endpoints – one that accepts temperature and one that accepts accelerometer data.

To create the advertising server, we built a small php server to serve three different advertisements, one legitimate, two malicious. The vulnerabilities shown in Figures 6 and 7 were embedded in the MetaWear app. The two malicious ads on the ads server were created to respectively exploit each vulnerability.

4.4. Collecting data with the audit hooks

After implementing the case study, we injected the audit hooks in every function which accessed any sensitive data, following the process described in Section 3.3. Next, we launched the advertisement server and the data aggregation service. Finally, we turned on the SecuWear server to allow the hooks to log their events for analysis.

To explore and capture data regarding normal operations (Figure 5), we launched the app on Android and told the advertising server to send a legitimate advertisement to the app. It began communicating with its advertisement server and its data aggregator server. We also launched the Ubertooth and its audit hook, from a physically proximally located device (i.e. a device in the same room as the wearable and the

phone). This began collection of Bluetooth network data and streaming audited events to SecuWear.

Next, we began gathering data for use in our analysis. Overall, we repeated each scenario (normal, code injection, denial of service) 25 times. In each case, we paired the app with the wearable. For the attack scenarios, we instructed the advertising server to serve up the appropriate malicious advertisement.

Henceforth, we refer to each repetition of the scenario as a *run*. Each run ran the scenario for approximately two minutes. During those two minutes, the audit hooks gathered data from each component in the system. The Ubertooth data was monitored in WireShark to confirm that a CONNECT_REQ packet [36] was captured after pairing. Connection requests are specific packets in the Bluetooth protocol that signal and initiate the pairing process. Since Bluetooth LE has 3 channels where this can occur, the Ubertooth does not always capture the packet [15]. This can be overcome by using three Uberteeth. Once connection was confirmed, we let the run continue for the two minutes. If a connection was not confirmed, the run was restarted. At the end of each run, captured events on SecuWear were arranged chronologically into event traces and output to a CSV file for analysis.

After capturing 25 runs of each scenario, for a total of 75 runs, we observed each run had an average of 669 events in their trace, with individual runs having event totals in the range 305-1,674. In total, this amounted to 50,249 events.

4.5. Feature Selection

To analyze the data and apply our chosen supervised machine learning techniques described in Section 3.4 we went through a series of processes: dataset cleaning and preparation, feature selection, division of the data into training and test sets, model creation, and classifier assessment analysis.

Our data cleaning process consisted of two steps: 1) removed malformed events (this only occurred in two of the ~45000 total events); 2) remove network traffic for other devices on the network (i.e. with mac addresses that do not match the phone or wearable).

To reduce dimensionality and complexity, we applied a mutual information feature selection criterion [37] to filter the total available features in the data to only those related to event and eventTypes. From the 75-run dataset, we extracted unique occurrences of various *eventtypes* and *events*. The eventtypes are categories of events that occurred during the various scenarios.

There are 17 different *eventtypes*, 2 from the BLE audit hook captured by Ubertooth – *ATT* and *LE LL*. ATT indicates a packet capture event with ATTribute data was received. LE LL is also a packet capture event, but occurs at a lower layer down from ATT. It usually relates to non-data transfer in Bluetooth. Both types relate to the BLE protocol used for connection and data exchange between MetaWear and Android app. The remaining 14 eventtypes include – *'Activity created', 'Activity destroyed', 'Blte server from MetaWewar binded', 'Chart updated', 'Creating user interface', 'Interaction with JavaScript', 'Item selected', 'MetaWear board prepared', 'Reset', 'Sensor initialized', 'Superclass constructor invoked', 'User interface created', 'axis setting configured', and 'x-axis configured to end'*. These types are specific to the Metawear Android app and relate to different functions being called as thermometer data was collected by MetaWear, displayed to the user, and sent to data aggregator web service.

There were 32 unique *events* which were grouped into the various *eventTypes* across the 75 runs. 11 among them were generated by the BLE channel across the *ATT* and *LE LL* eventTypes. Among the 11 events from BLE channel *'ADV_IND'* and *'ADV_DIRECT_IND'* were connectable undirected and directed advertising events created by the MetaWear device. '*ADV_NONCONN_IND'* was a non-connectable undirected advertising event created by MetaWear. *'ADV_SCAN_IND'* was a scannable undirected advertising event which indicates a device cannot be connected to but can respond scan requests. '*SCAN_REQ'* and *'SCAN_RSP'* were scan requests sent by the Android device and scan response events sent back by the MetaWear device. *'CONNECT_REQ'* was a connection request event sent by the Android device to connect to MetaWear. The other four were *LLID:0* to *LLID:3* were events which occurred during data exchange between MetaWear and Android and indicate that Bluetooth variable reads/writes.

The remaining 21 events were generated by the Android app. Each was of the form *'<foo> method executed'* indicating the execution of method *foo* at some point during the run. Within this group, we can classify the 21 events into two classes: 1) interaction with the JavaScript Interface (e.g. by a webview), and 2) execution of methods for connection and data exchange (e.g. from the MetaWear or to the aggregating web service).

4.6. Dataset division and Model creation

We created two models, one based on eventType categories and one based on events. We set aside 80% of our data for training and reserved 20% for testing. To prevent sampling bias, we randomly applied the 80/20 sampling categorically across the three types of

runs, i.e. 80/20 code injections, 80/20 denial of service, and 80/20 normal data. This ensured a representative sample of each categorically group was present in both the training and test data. Using the sampled training data, we trained a model using the event features and a second on the eventType features.

4.7. Case study: Results and discussion

Table 1 contains the average precision and recall for each ML technique applied over the eventType feature set. Similarly, Table 2 contains the average precision and recall for each ML technique applied to the event features. "Standard" definitions of precision and recall are used in both tables, defined as follows:

$$\text{Precision} = TP / (TP + FP)$$
$$\text{Recall} = TP / (TP + FN)$$

where *TP* is a true positive, *FP* is a false positive, and *FN* is a false negative. To compute average precision and recall we ran 40 trials using stratified random sampling to sample from the 75 run dataset – training each algorithm on the dataset each time and then using the test set to compute precision and recall. Our experiment shows that SVM worked best for attack detection in both datasets, while Naïve Bayes performed poorly, and the other techniques fell somewhere in between. It should be noted that since the model is fine grained, i.e. trained to recognize different attacks not just attack/non-attack, precision and recall are fine-grained as well.

Table 1. Average precision obtained from various classification models on dataset with frequency of eventTypes

	Logistic regression	SVM	Naïve Bayes	K-NN
Average precision	81%	86%	59%	72%
Average Recall	80%	85%	60%	67%

Table 2. Average precision obtained from various classification models on dataset with frequency of events

	Logistic regression	SVM	Naïve Bayes	K-NN
Average precision	73%	88%	67%	75%
Average Recall	73%	83%	67%	68%

While the specifics of these attack scenarios and the efficacy of the ML techniques for attack detection are interesting, **the results of our study lend support for the use of audit hooks towards capturing relevant application data to support security investigations.**

The high precision and recall suggest that our approach could be useful as "one more tool in the shed" for dynamic security testing. This claim is, as noted in our methodology, limited to this case. Generalizing it to larger and other cases may be possible, but would involve additional study – as we note later in the limitation discussion below.

Another question we wanted to explore was what size of data would be required to get results in our case. To answer this question, we examined the relationship between training size and false positive / false negative rates to identify the minimum threshold of sample size required to train the SVM model and obtain acceptably stable results. Measuring this, we varied the size of the training set used to train the SVM model and, for each size, computed the average false positive and negative rates. Figure 8 display these results graphically. From the graph we can see that, at least for this data, training set sizes with at least 30 runs provided moderate results, but larger sizes are certainly desirable. From this data, it is reasonable to suggest that the criticality of the software/system under test could reasonably govern the choice of training set size. For low criticality systems, a smaller training set (less data collection time) might be justifiable. The results show that the data obtained by audit hooks can be efficiently used to generate models and classify attacks from non-attacks. Almost all attack scenarios are detected, with average precision of no less than 86%. It should be noted that although our model was trained after the various data capture runs, the model could operate in real-time to detect and report threats as they emerge.

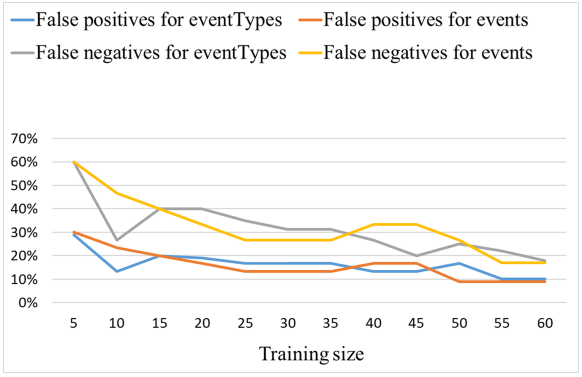

Figure 8: False positive and false negative rates vs training set size in SVM

5. Limitations and conclusion

Our multi-component security analysis audit hooks provide knowledge that spans component boundaries. While this can be illustrative and helpful for identifying component-spanning attack vectors and vulnerabilities, it comes at a cost. That cost is the need to insert audit hooks in each function of interest. This constitutes a non-trivial amount of DevOps time. This limitation could be mitigated with tool support that encourages developers to insert audit hooks during the development process – instead of as an afterthought.

In addition to this, our results are limited by the number of cases studied, the number of attacks explored, and the size of the dataset considered in the evaluation. These factors suggest that the reader should be careful not to overly generalize the results. Taken for what it is, the case study provides a proof-of-concept that the data captured by audit hooks in IoT systems can be used for ML and classification in support of security analysis and weakness mitigation.

Future work will focus on generalizability to explore how well the results hold for other cases and contexts. We also look forward to using the audit hooks to support additional helpful features for security evaluation such as mining collected event data to isolate vulnerable components and make mitigation suggestions for resolving potential security issues.

6. References

[1] T. Yilmaz, R. Foster, and Y. Hao, "Detecting vital signs with wearablewireless sensors," *Sensors*, vol. 10, no. 12, pp. 10837–10862, 2010.

[2] M. Patel and J. Wang, "Applications, challenges, and prospective in emerging body area networking technologies," *IEEE Wirel. Commun.*, vol. 17, no. 1, pp. 80–88, 2010.

[3] A. Hern, "Fitness tracking app Strava gives away location of secret US army bases," *The Guardian*, 2018.

[4] "Understanding the Internet of Things (IoT)," *GSMA Assc.*, 2014.

[5] D. Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J.A., Invernizzi, L., Kallitsis, M. and Kumar, "Understanding the mirai botnet," in *USENIX Security Symposium*, 2017.

[6] E. Bertino and N. Islam, "Botnets and Internet of Things Security," *Computer (Long. Beach. Calif).*, vol. 50, no. 2, pp. 76–79, 2017.

[7] H. O. Sullivan, "Security Vulnerabilities of Bluetooth Low Energy Technology (BLE)," *Tufts University*, 2015.

[8] U. Lamping, R. Sharpe, and E. Warnicke, *Wireshark User's Guide*, 2014.

[9] Gordon Fyodor Lyon, *Nmap Network Scanning: The Official Nmap Proj. Guide to Network Discov. and Sec. Scanning*, 2009.

[10] "Metasploit | Penetration Testing Software, Pen Testing Security | Metasploit." [Online]. Available: https://www.metasploit.com/. [Accessed: 22-May-2018].

[11] M. L. Hale, D. Ellis, R. Gamble, C. Waler, and J. Lin, "Secu Wear: An Open Source, Multi-component Hardware/Software Platform for Exploring Wearable Security," in *Proceedings - 2015 IEEE 3rd Int'l Con. on Mobile Services, MS 2015*, 2015.

[12] S. Siboni, A. Shabtai, N. O. Tippenhauer, J. Lee, and Y. Elovici, "Advanced Security Testbed Framework for Wearable IoT Devices," *ACM Trans. Internet Tech.*, vol. 16, no. 4, pp. 1–25, 2016.

[13] J. Liu and W. Sun, "Smart Attacks against Intelligent Wearables in People-Centric Internet of Things," *IEEE Commun. Mag.*, vol. 54, no. 12, pp. 44–49, 2016.

[14] M. Lee, K. Lee, J. Shim, S.-J. Cho, and J. Choi, "Security Threat on Wearable Services: Empirical Study using a Commercial Smartband." 2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia), 2016.

[15] M. Ryan, "Bluetooth: With Low Energy Comes Low Security," *Proc. 7th USENIX Conf. Offensive Technol.*, p. 4, 2013.

[16] MITRE, "CVE List cpe:/o:google:android:6.0.1," 2017.

[17] M. Gegick and L. Williams, "Matching attack patterns to security vulnerabilities in software-intensive system designs," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, p. 1, 2005.

[18] J. West, T. Kohno, D. Lindsay, and J. Sechman, "WearFit: Security Design Analysis of a Wearable Fitness Tracker," *IEEE Cyber Secur.*, 2016.

[19] S. Anwar *et al.*, "From intrusion detection to an intrusion response system: Fundamentals, requirements, and future directions," *Algorithms*, vol. 10, no. 2, 2017.

[20] Y. Zhang, W. Lee, and Y.-A. Huang, "Intrusion detection techniques for mobile wireless networks," *Wirel. Networks*, pp. 545–556, 2003.

[21] D. M. Best, A. Endert, and D. Kidwell, "7 Key Challenges for Visualization in Cyber Network Defense," *Proc. Elev. Work. Vis. Cyber Secur. - VizSec '14*, pp. 33–40, 2014.

[22] W. A. Jansen, "Cloud Hooks: Security and Privacy Issues in Cloud Computing," in *44th IEEE Hawaii International Conference on System Sciences*, 2011.

[23] J. Schütte and G. S. Brost, "LUCON: Data Flow Control for Message-Based IoT Systems," *arXiv preprint* 1805.05887, 2018.

[24] A. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Commun. Surv. Tutorials*, vol. PP, no. 99, p. 1, 2015.

[25] "PyShark." [Online]. Available: https://kiminewt.github.io/pyshark/. [Accessed: 12-May-2018].

[26] "square/okhttp." [Online]. Available: https://github.com/square/okhttp/. [Accessed: 02-Dec-2017].

[27] "JSON API — A specification for building APIs in JSON." [Online]. Available: http://jsonapi.org/. [Accessed: 18-Apr-2018].

[28] "Httplib2." [Online]. Available: https://github.com/httplib2/httplib2. [Accessed: 02-Dec-2017].

[29] "The Web framework for perfectionists with deadlines | Django." [Online]. Available: https://www.djangoproject.com/. [Accessed: 15-May-2018].

[30] N. Nadusumilli, M. L. Hale, and I. Shrestha, "SecuWear Client Ubertooth." [Online]. Available: https://github.com/MLHale/secuwear-client-ubertooth. [Accessed: 20-May-2018].

[31] I. Shrestha and M. L. Hale, "MetaWear Android Sample App." [Online]. Available: https://github.com/IsaacShrestha/MetaWear-SampleApp-Android. [Accessed: 20-May-2018].

[32] I. Shrestha, "MetaWear sample web app." [Online]. Available: https://github.com/IsaacShrestha/secuwear-webapp. [Accessed: 20-May-2015].

[33] "MetaWearC Streaming Sensor w/ 6 Axis IMU + Temp - MbientLab." [Online]. Available: https://mbientlab.com/product/metawearc/. [Accessed: 11-Nov-2017].

[34] "WebView | Android Developers." [Online]. Available: https://developer.android.com/reference/android/webkit/WebView. [Accessed: 25-May-2018].

[35] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android System," in *Proceedings of the 27th ACM Computer Security Applications Conference*, 2011.

[36] TI, "CC26x0 SimpleLink Bluetooth Low Energy Software Stack 2.2.x Developer's Guide," 2010.

[37] H. Peng, F. Long, and C. Ding, "Feature selection based on mutual information: Criteria of Max-Dependency, Max-Relevance, and Min-Redundancy," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 8, pp. 1226–1238, 2005.